μp - L - 01

(A) μp — in PC, mobile, remote → cause there is programming
— not in fan, life → no programming

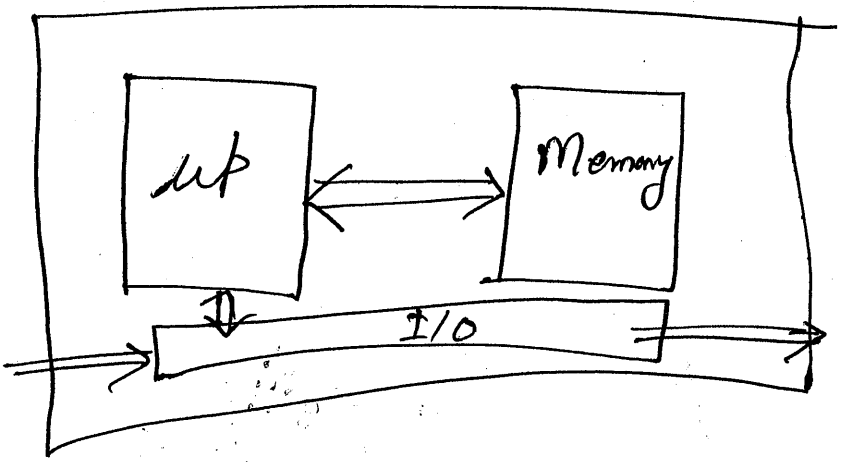
executes the program

(A) Why not study i-7? Does a traffic light have a cori-7?

(A) progress: 8085 → 8086 → 80186
286
386
⋮

(A) μp in PC → CPU → inside the fan

Computer System



⊛ In 8086, memory is RAM & ROM, not the secondary memories.

floppy disc
CD

⊛ Memory stores data ← image, song, movie

↓

how? series of 0s and 1s.

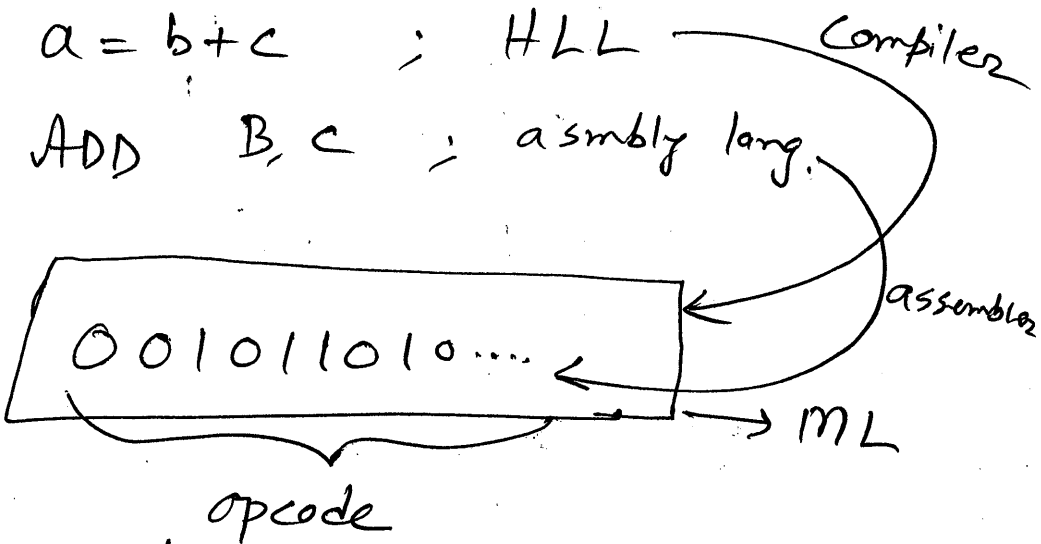
⊛ Instruction cycle:

1) fetch instn. from memory

2) Decode:

fetched instn. is 0s and 1s.

understanding the 0s and 1s.



3) Executing

Few basics

Dec	Hex	Bin
0	0H	0
⋮	⋮	⋮
9	9H	1001
	A	
	⋮	
	F	

µp uses Hex. system cause more information can be stored.

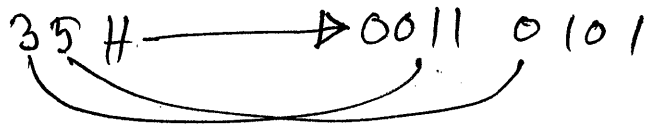
ex: for a 4-bit system,

decimal max 9999

hex max FFFF

so, more info. in hex system.

Hex to Bin



8 bit nums	16 bit nums
00 H	0000 H
⋮	⋮
FF H	FFFF H

Power of 2

$2^{10} = 1k$

$2^{11} = 2 \times 1k = 2k$

$2^{12} = 2^2 \times 1k = 4k$

$2^{13} = 2^3 \times 1k = 8k$

$2^{20} = 1K \times 1k = 1M$

$2^{24} = 2^4 \times 1M = 16M$

$2^{30} = 2\ 1k \times 1k \times 1k = 1G$

$2^{40} = 1T$

$2^{32} = 4G$

Bus

Addr. bus
Data bus
Control bus.

} transfers 0s and 1s

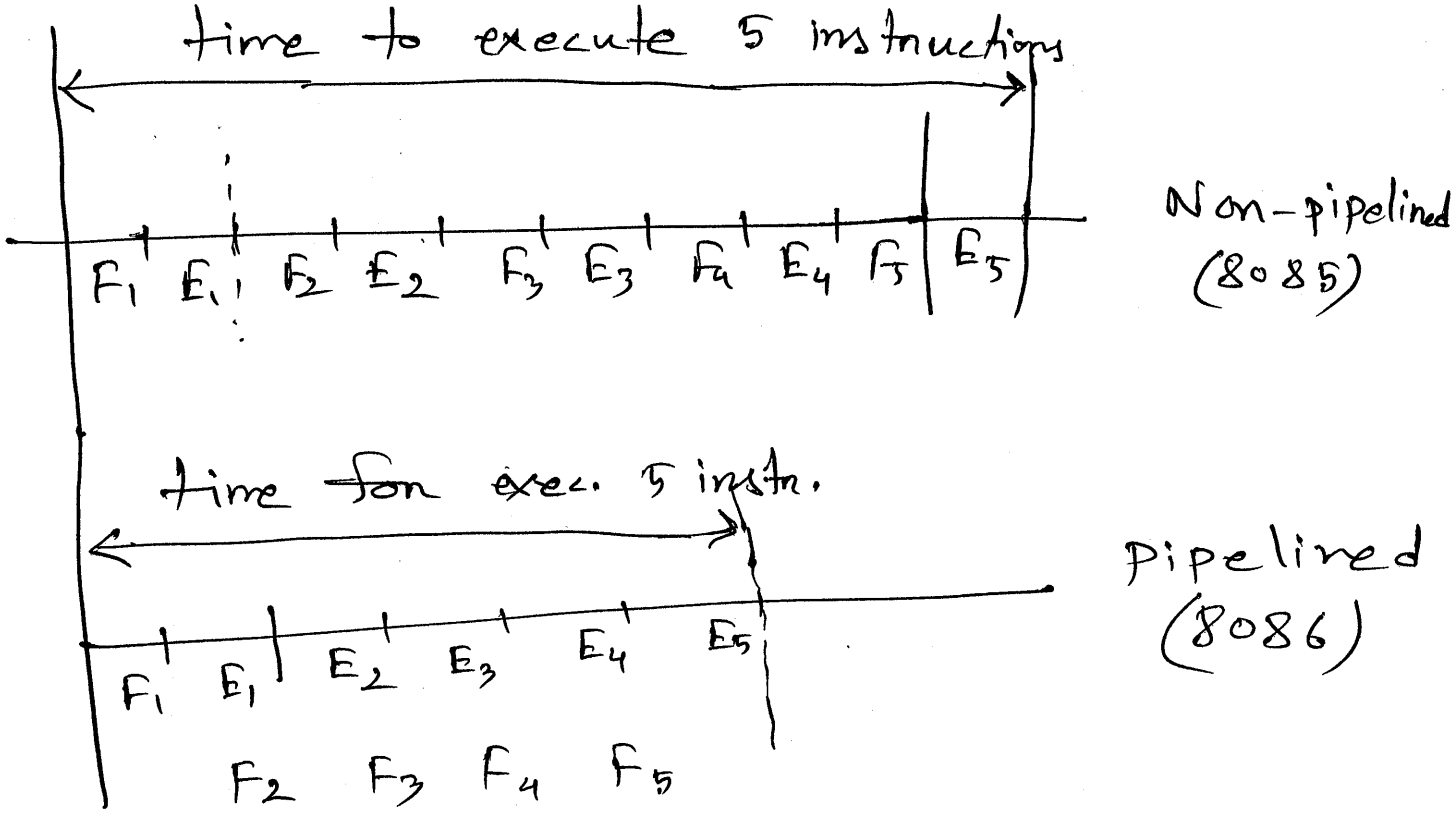
4 bit lines — 4 lines in bus

8 " " — 8 " "

L-02

Pipelining

- What makes up faster? why 2.6 GHz, 3.6 GHz these days?
- Ans is Pipelining
- 8086 → 1st processor to introduce pipelining



- that's why - 2 units in 8086 architecture : one for Fetching, one for executing

— this is 2 staged pipelining

Disadvantages :

1) If a instruction needs its previous one's output for execution, it can't start until the previous one is finished.

2) Branching : If instn. 1 is execution tells us to go to instruction 8, and we fetch instn. 2 meanwhile, it has to be discarded. (ex: if-else)

Branch issue solution :

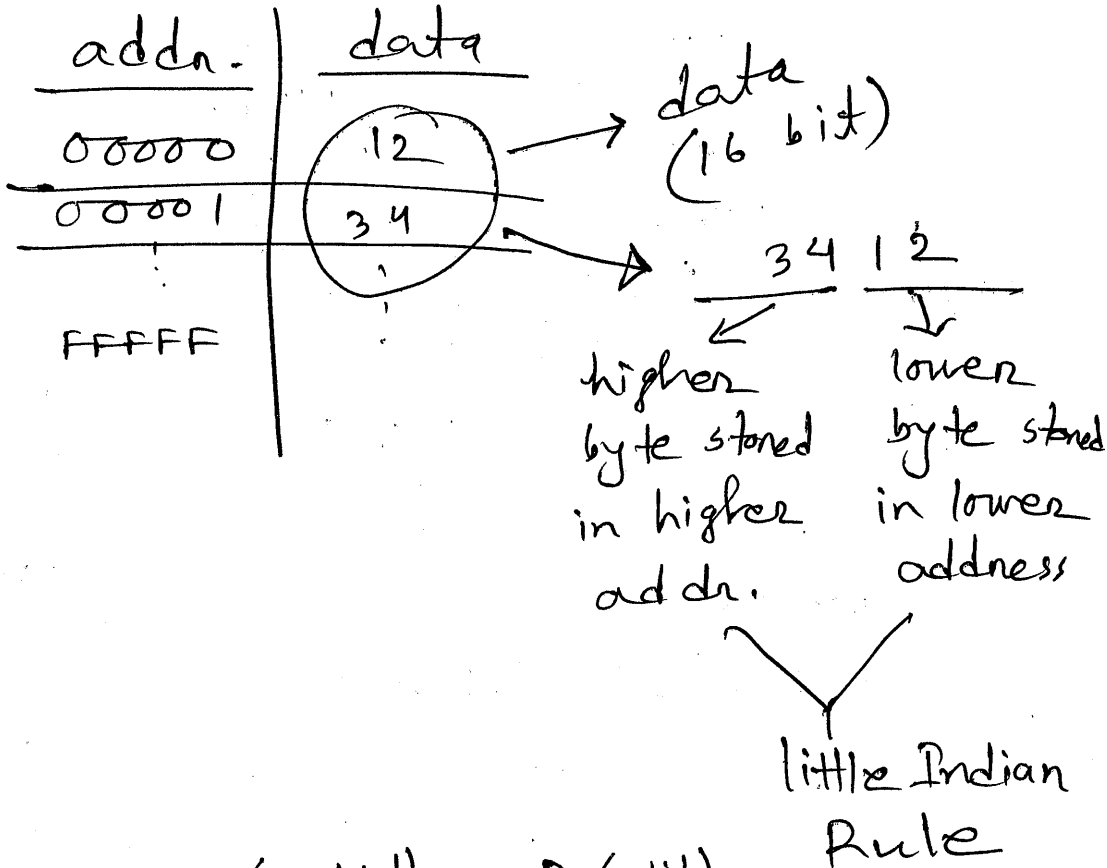
- Branch prediction algorithm
- decides whether it will branch or not based on previous many decisions.

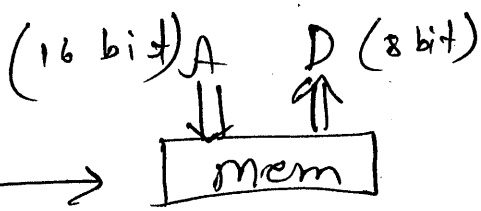
(8086 doesn't implement it)

8086 memory banking

- 20 bit address bus
- so, memory size = 2^{20} = 1M
- 1 memory location contains 1 byte of data

ex:

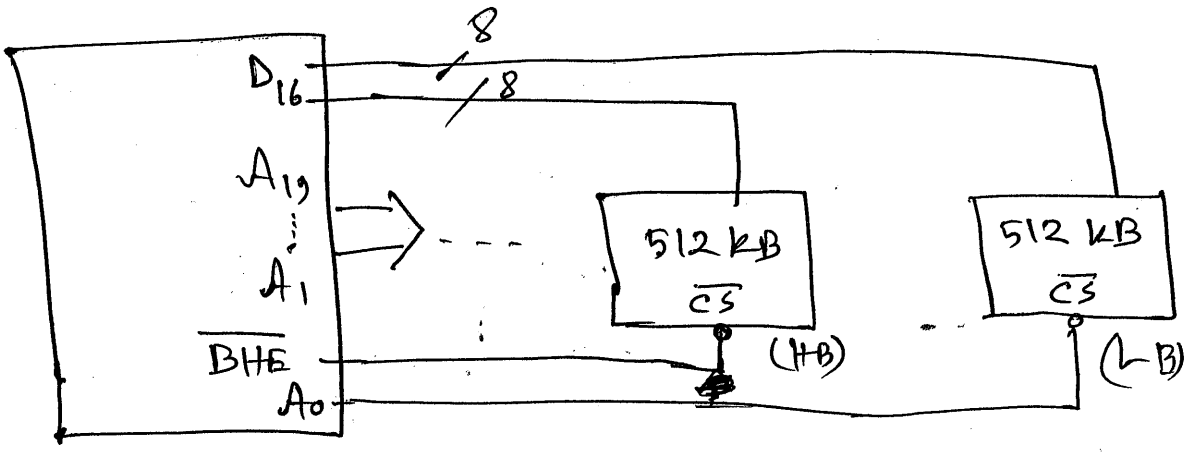




- problem? → we want to transfer this data 3412 in one cycle. But up can't generate two addresses in one cycle.

- ~~Stupid soln~~:

- Soln: We want to access 16 bit i.e. 2 bytes of data in a cycle. So, have 2 memory chips.



1 MB

512 kB
Odd bank (higher bank)
00001H
00003H
00005H
00007H
FFFFF

512 B
even bank (lower bank)
00000H
00002H
00004H
00006H
FFFFE

- We can choose addr 00000H and 00001H together. So do we can 00002H and 00003H.

- A_0 selects which bank to choose
- $A_1 - A_{19}$ selects the memory location. Same mem. location is selected in both banks. That's why 00001H and 00002H aren't selected together.
- $A_0 = 0$; LB selected

  $A_0 = 1$; LB not selected

  $\overline{BHE} = 0$; HB selected

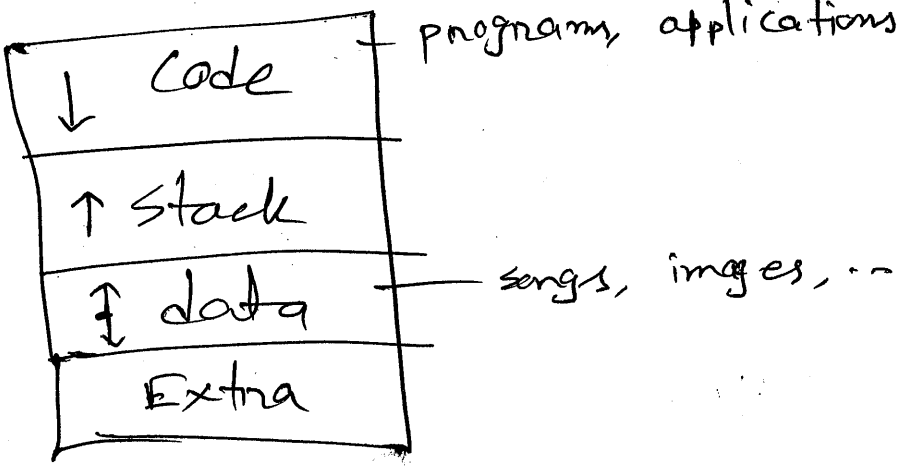
$\overline{BHE}$	A_0	operation
0	0	R/W 16 bit from both banks
0	1	R/W 8 bit from HB
1	0	R/W 8 bit from LB
1	1	None (idle)

that's the benefit of banking.
We can access either LB/HB/both.

L-03

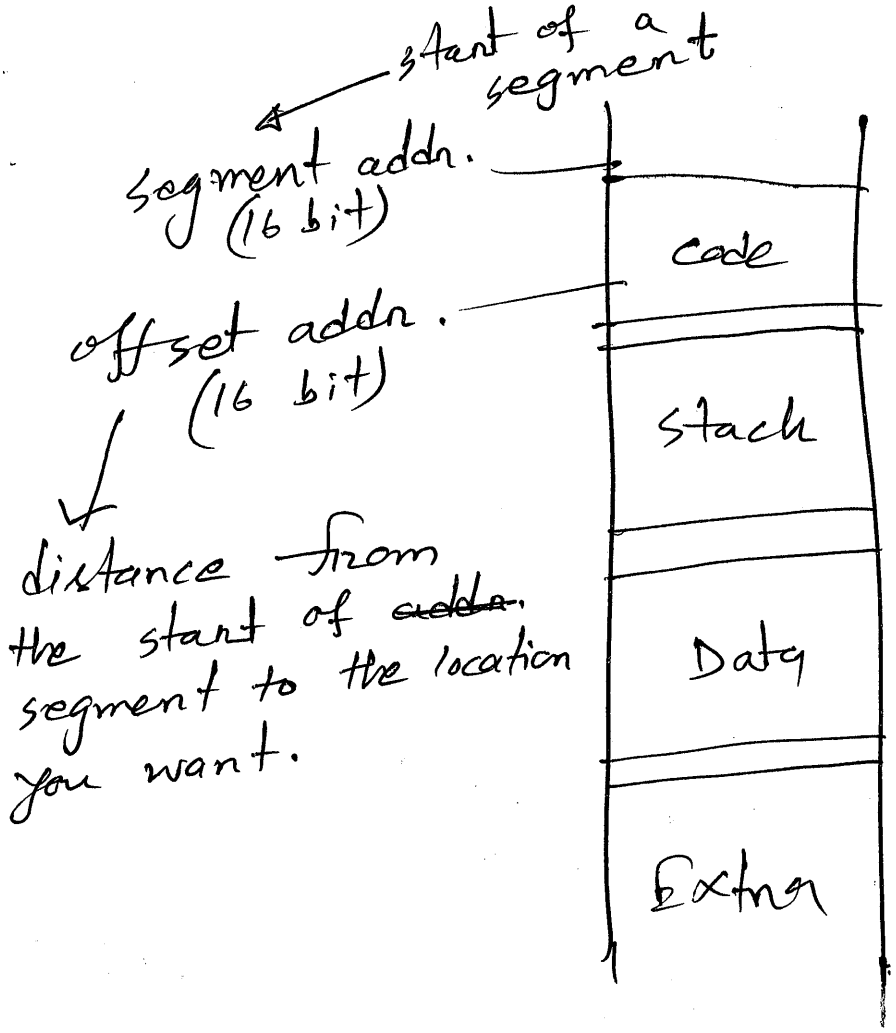
Memory Segmentation

- 4 segments



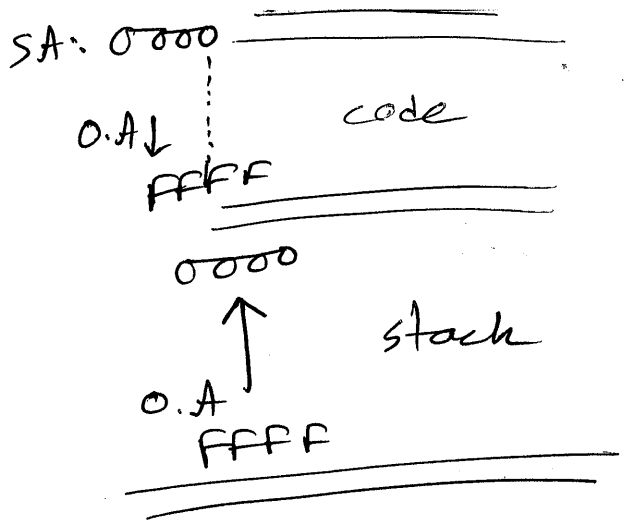
- this is the birth of concept of files. In 1970, no computer had files & folders. But files evolved and file is basically a modern version of segments.

- 1 MB memory ≡ 20 bit address this 20 bit addr. is physical addr. or actual addr. But, while programming, we need a byte compatible address / 16-bit addr.



- every time, u don't give both SA and offset. you give SA at first, then give only offset address.
- 8085 didn't have segmentation. Why? (cause 16-bit address)
- Why segments don't overwrite? cause, if you keep increasing the offset addr., it will reach FFFF at one point and u can't go further. If you could, there'd

be overwrite.



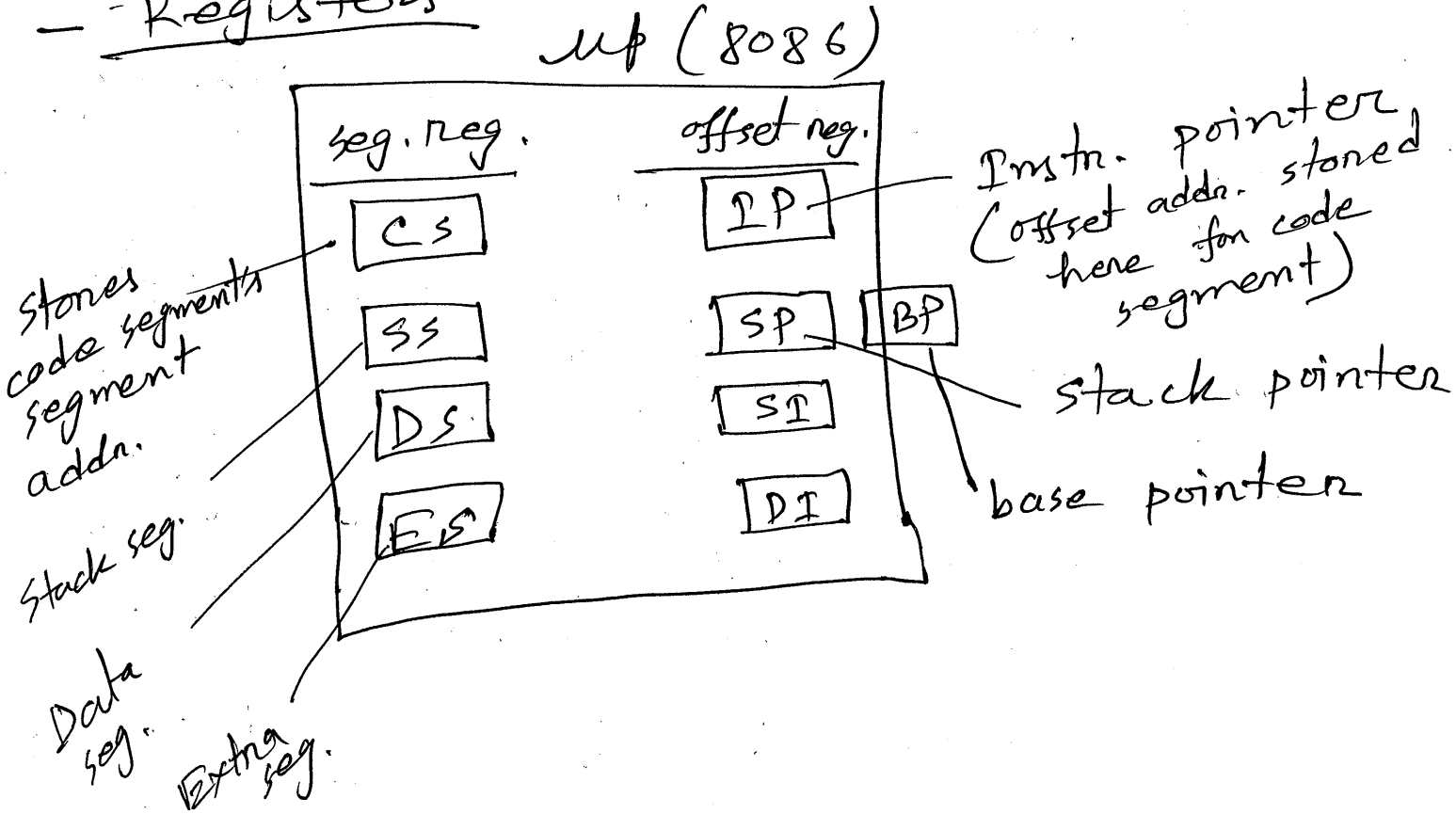
- max. range of a segment

0000H → FFFFH

max size of segment = 2^{16} = 64k

(Not all segments are 64k)

- Registers



In short: CS — stores code segment's segment addn.

IP — code segment offset address.

- you give up the segment addn. and offset addn. It's the job of up to convert them to physical address cause up will send PA by 20-bit addn. bus to mem.

- PA = seg addn. × 10 + offset } up does this calculation
  = CS × 10 + IP

- If SS = 3000H, from what physical address stack segment starts?
  = 30000H

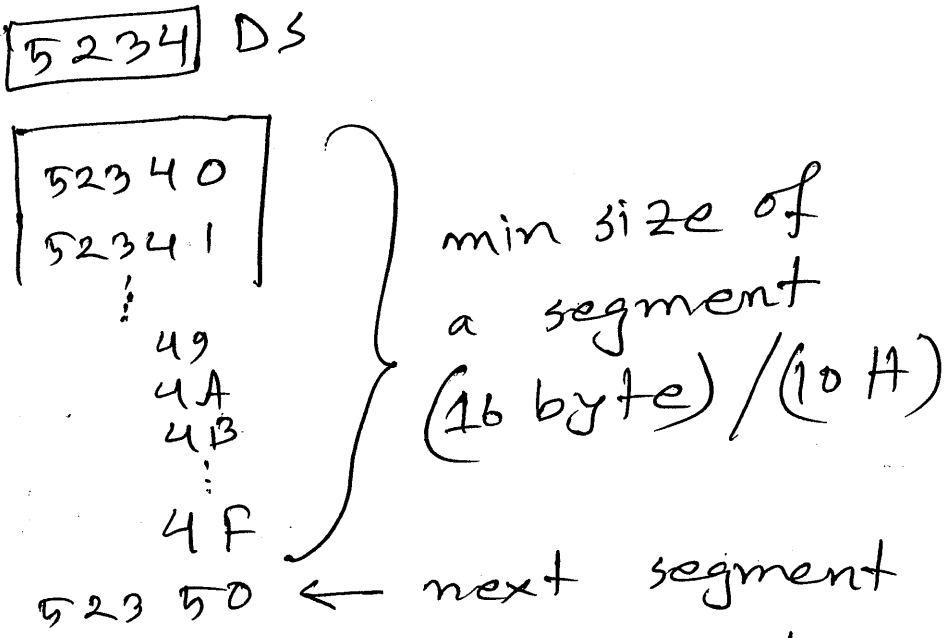
- want to start stack segment at → 52350 → SS = 5235

31430 → 3143

52945 → 22 ... (possible)

NO! need to be 10's multiple

- so, if a segment ends at 52340, the next segment can start at 52350. How?



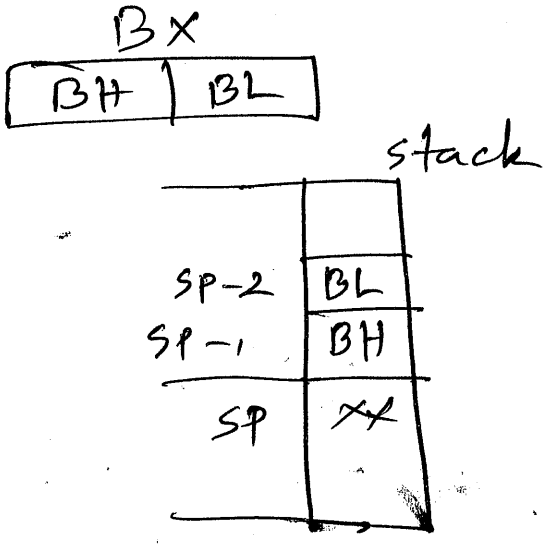
cause the next segment can't start at 52341 or 52342, ... need to be 10's multiple.

- SP and BP

- after push and pop, SP ↑ or ↓
  SP points to top of stack.
  but BP is used for random access of stack.
- BP can point anywhere on stack.
- ex : incoming sms in phone.

- Push , POP

push BX

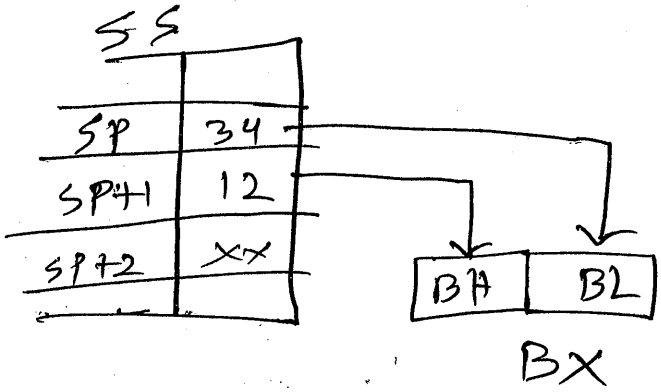


SS : [SP-1] ← BH

SS : [SP-2] ← BL

SP ← SP-2

POP BX



BL ← SS : [SP]

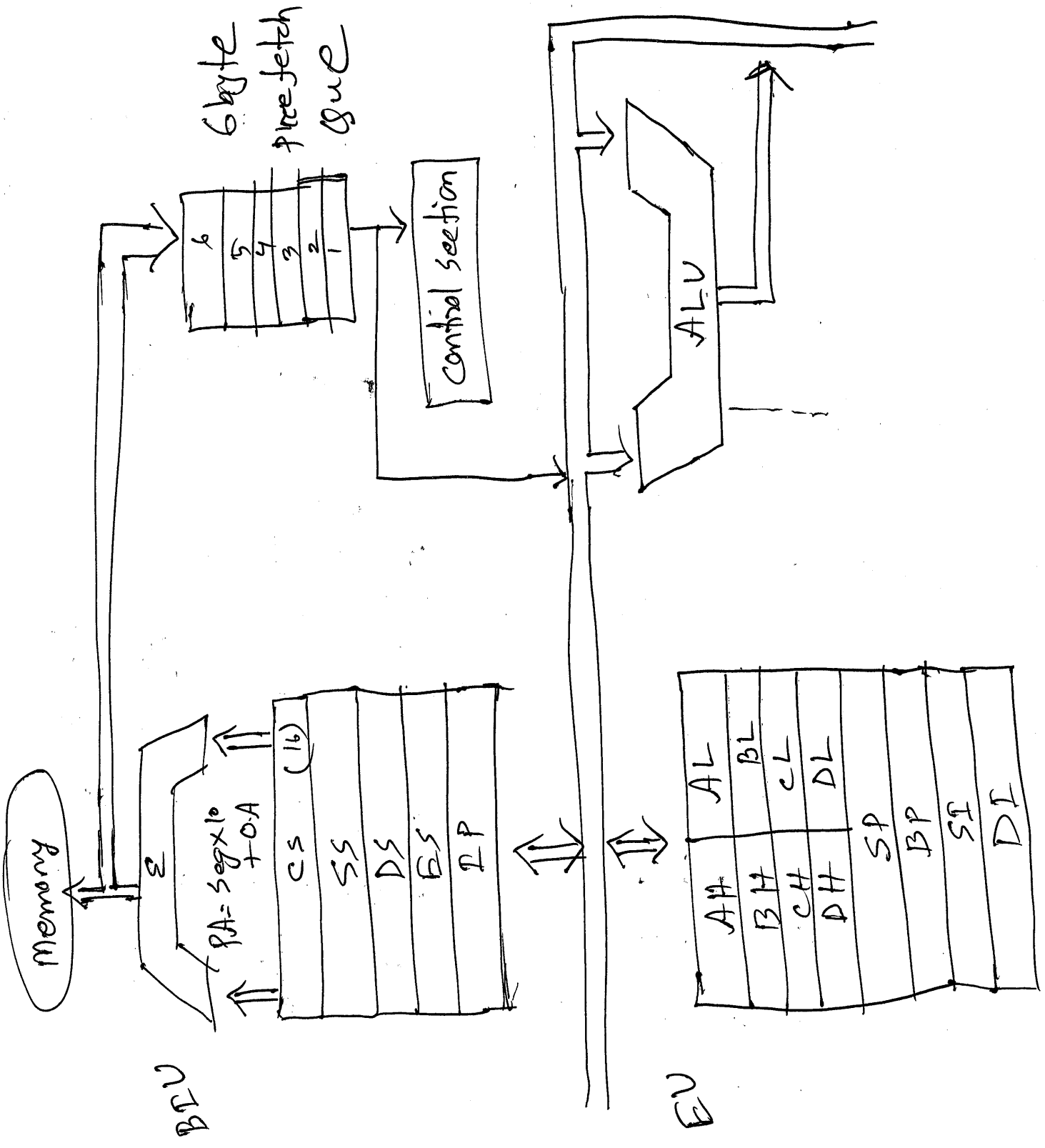
BH ← SS : [SP+1]

SP ← SP + 2

L-04

Internal Architecture

(A) : Arithmetic circuit



- Instn. Q is of 6 bytes, not 6 instn. instructions can be of different sizes.
- when 2 bytes location in Q is free, next 2 bytes of instructions are fetched. If
- 1 byte is free, nothing will happen.
- If a half of instn. comes within 2 bytes, any problem?
  - No, cause when it reaches bottom of Q, it will be a full instructions.

- control section → decodes the instruction

for `Mov BL, 04H`

control section triggers BL to load value 04H

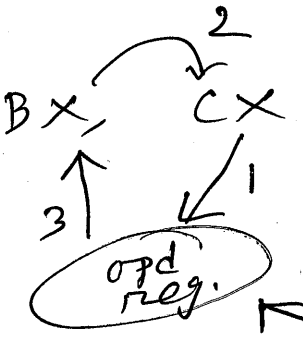
for `Add BL, CL`

control section sends add signal to ALU

(*) operand registers (16 bit)

flag registers (16 bit)

XCHG



temporary registers

Flags Registers

(*) Use slide

(*) sign bit — MSB → sign flag

- 8 bit unsigned numbers

  0 — 255

- 8 bit signed numbers

  Dec: −128 ---- 127

  Hex: neg: − 80H ------ 01H (1000 0000 ... 1111 1111 = FF)

  pos: 00H ------ 7FH (0000 0000 ... 0111 1111)

④ sign flag gives wrong sign in case of overflow.

If OF = 1, sign flag is wrong

So, never directly check the sign flag.

ex:

```
  42H      0100 0010
+ 23H      0010 0011
-----    ------------
  65H      0110 0101
```

OF	SF	ZF	AC
0	0	0	0
PF		CF	
1		0	

```
  37H      0011 0111
+ 29H      0010 1001
-----    ------------
  60H      0110 0000
```

OF	SF	ZF	AC	PF	CF
0	0	0	1	1	0

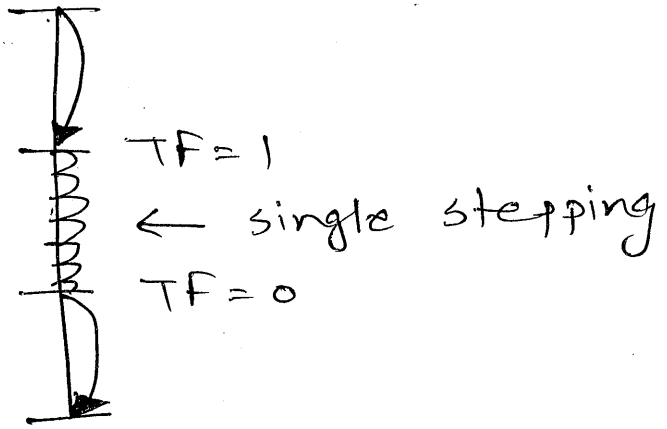
```
  42H      0100 0010
+ 43H      0100 0011
-----    ------------
  85H      1000 0101
```

OF	SF	ZF	AC	PF	CF
1	1	0	0	0	0

④ Control flags :

- controlled by us

TF : trap flag



IF = Interrupt flag :

1 : intrpt. enabled

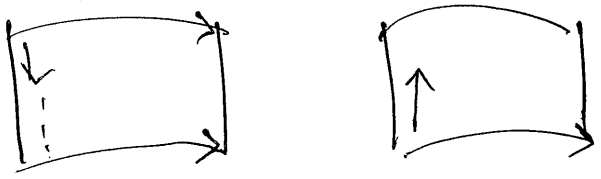
0 : intrpt. disabled

DF : Direction flag

1 : auto dec

0 : auto inc

ex: string copy paste



L-05

Addressing modes

: manner in which operand is given

1) Immediate —(data in instn.)→ ex: mov CL, 34H

2) Register —(data in reg.)→ ex: mov CL, BL
INC BX

3) Direct —(addr. in instn.)→ ex: mov CL, [2000H]:
CL ← DS:[2000H]
mov CX, [2000H]:
CL ← DS:[2000H]
CH ← DS:[2001H]
mov [2001H], CL
DS:[2001H] ← CL

4) Indirect —(address in reg)→
up takes addr. from registers, then use that address to fetch data
so indirect.

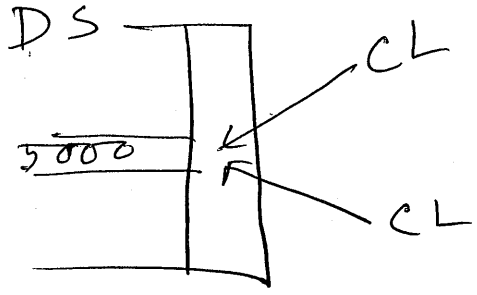
a) Register Indirect:

mov CL, [BX] ; CL ← DS:[BX]

mov CX, [BX] ; CL ← DS:[BX]
CH ← DS:[BX+1]

Classwork

(1)



direct

mov CL, [5000H]

indirect

mov BX, 5000H
mov CL, [BX]

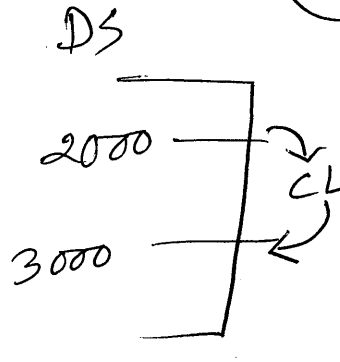
advantage:

mov CL, [BX]
INC BX

Accessing series of location

in a loop, we can read data from 5000H then from 5001H, ...

(2)

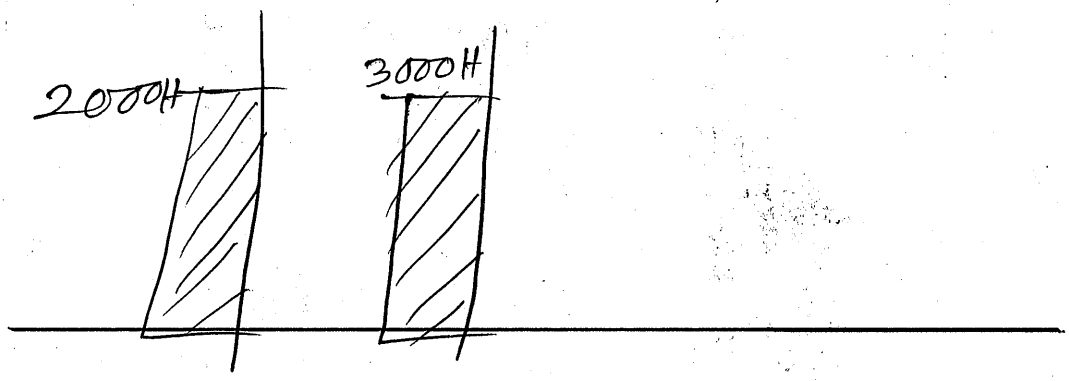


take data from location 2000H and put in 3000H.

Ans:

mov CL, [2000H]
mov [3000H], CL

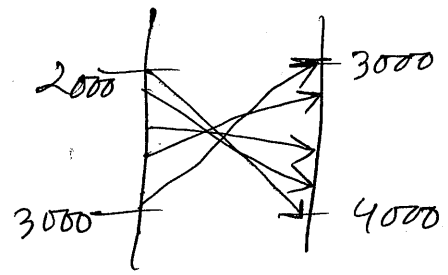
③ block transfer program :




```
mov   SI, 2000H
mov   DI  3000H
(mov  CL, [SI]
 mov  [DI], CL   ) loop
 INC  SI
 INC  DI
```

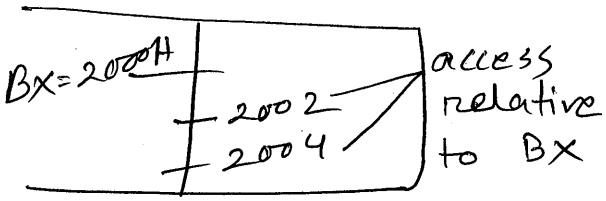
④ block inversion :

```
mov   SI, 2000H
mov   DI, 4000H
(mov  CL, [SI]
 mov  [DI], CL
 INC  SI
 DEC  DI
```



b) Register Relative (addr ← reg + displacement)

mov CL, [BX + 03H]



CL ← DS[BX + 03H]

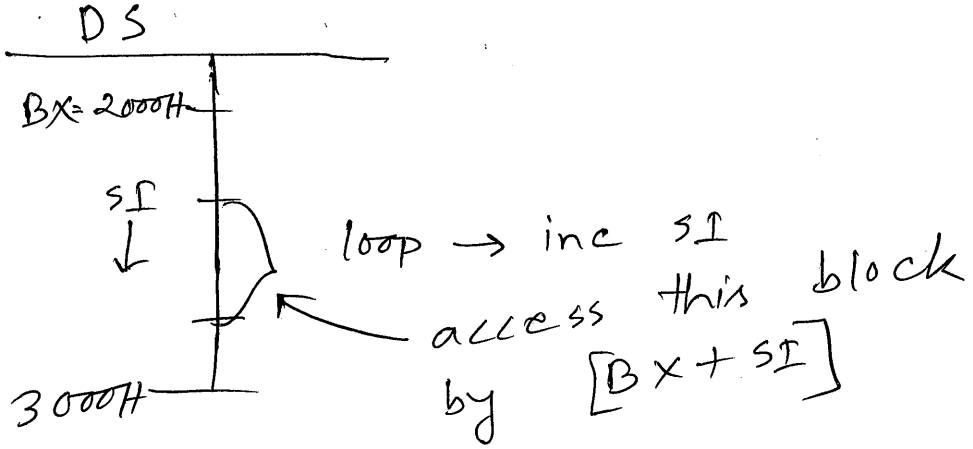
c) Base indexed (addr ← base reg. + index reg)

mov CL, [BX + SI]

CL ← DS:[BX + SI]

mov CL, [BP + SI]

CL ← SS:[BX + SI]



d) Base relative plus indexed :

addr ← base reg + idx reg + displacement

mov CL, [BX+SI+03H]

CL ← DS:[BX+SI+03H]

mov CL, [BP+SI+03H]

CL ← SS:[BX+SI+03H]

5) Implied — operand is implied

we give nothing, some instructions are meant for some operands

ex: STC ; set the carry flag
cf ← 1

CLC ; cf ← 0

~~DAA~~

L-06

8086 Minimum Mode

- $MN/\overline{MX}$ pin

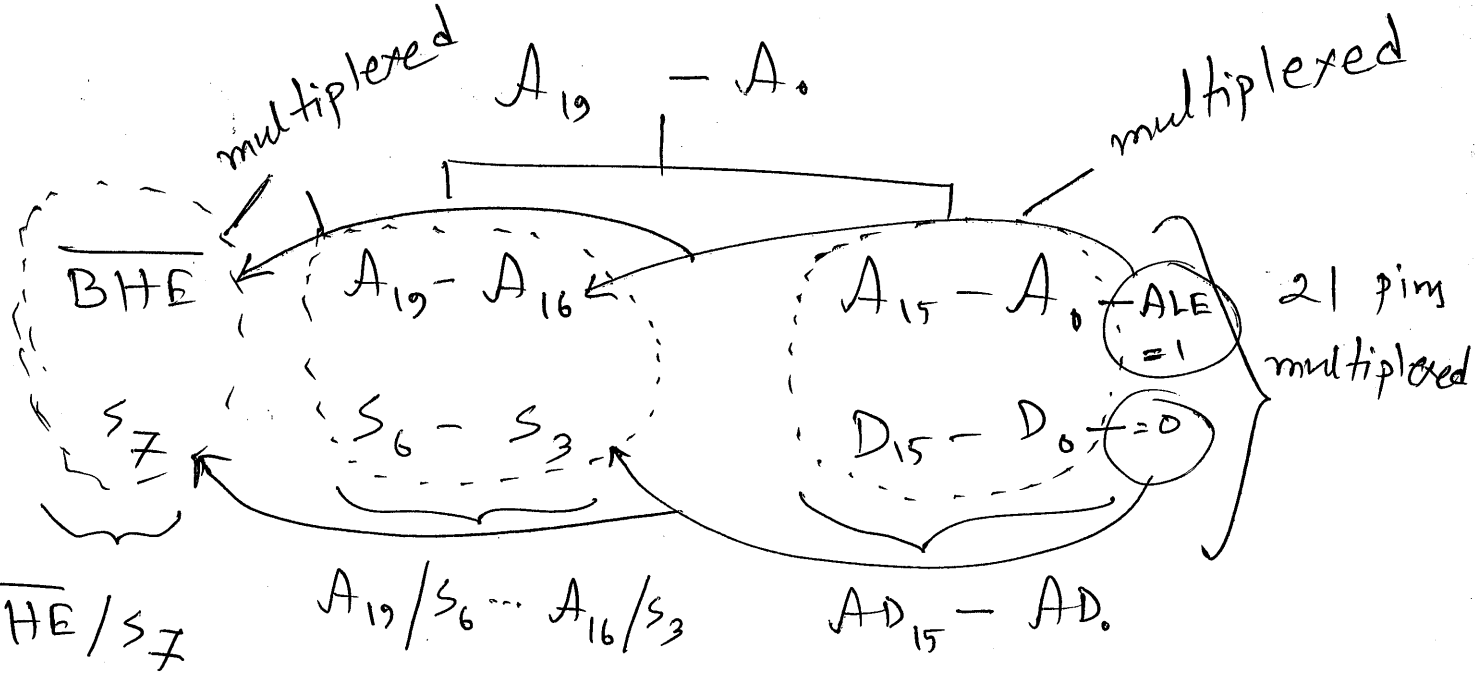
  1 → min mode

  0 → max mode

- min mode – one processor

  max mode – multiple "

- Multiplexing -



if ALE = 0, $AD_{15} - AD_0$ carries data

1, _____ address

S_4	S_3	
0	0	ES
0	1	SS
1	0	CS
1	1	DS

$S_5 \Rightarrow 0 \Rightarrow$ If $\Rightarrow 0$

1 $\Rightarrow$ If = 1

$S_6 \Rightarrow 0 \Rightarrow$ 8086 Bus Master

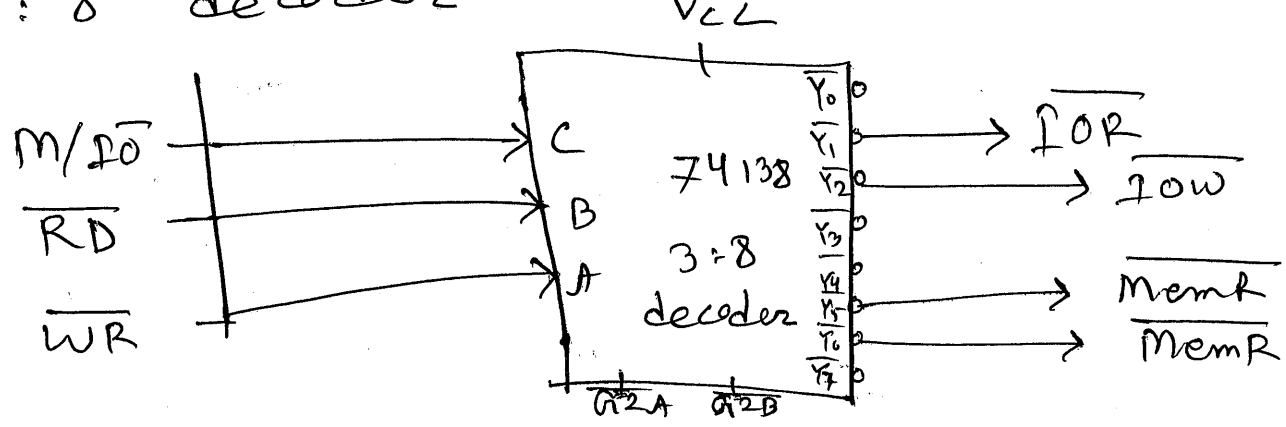
1 $\Rightarrow$ Other BM

So, in min mode, S_6 will be always 0, max mode 1.

(*)

M/$\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Op
0	0	1	IOR
0	1	0	IOW
1	0	0	Mem R
1	1	0	Mem W

(*) generating these signals using 3:8 decoder

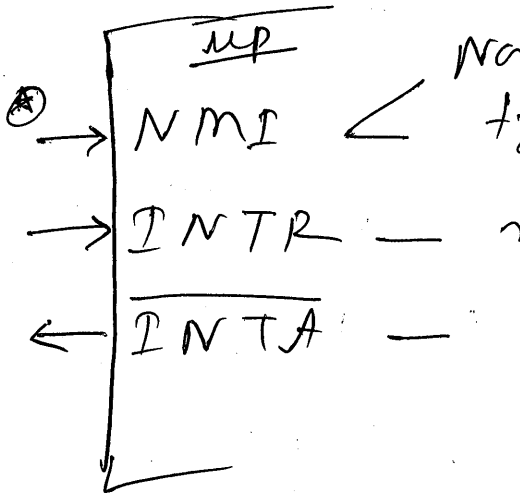


(*)

Clock :

clock is a tick to the up to change its state.

8086 works at 6MHz clk cycle. We've to give 6MHz clock to its clk pin with 33% duty cycle.

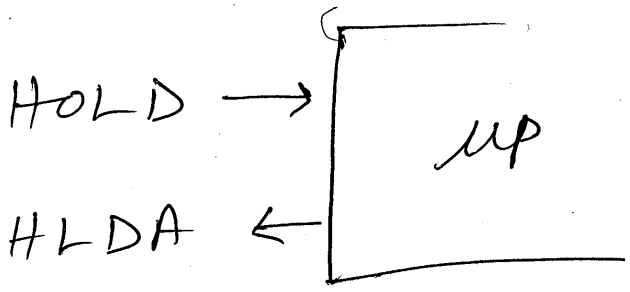


(*) NMI ← Non-maskable type 2 intrpt initiated

INTR — maskable, request a h/w intrpt.

INTA — up wants a intrpt vector number by sending intrpt ack.

(*)



by default, up is the BM. DMA Controller sets HOLD = 1 and sends HLDA to DMAC, up loses control over bus. DMAC becomes new BM and controls the data transfer betn mem. and IO. If DMAC sets HOLD=0

again, up gets control

(A) $DT/\overline{R}$: 1 → data transmit
0 → data receive

(B) $\overline{DEN}$: enable a transreceiver connected to up.

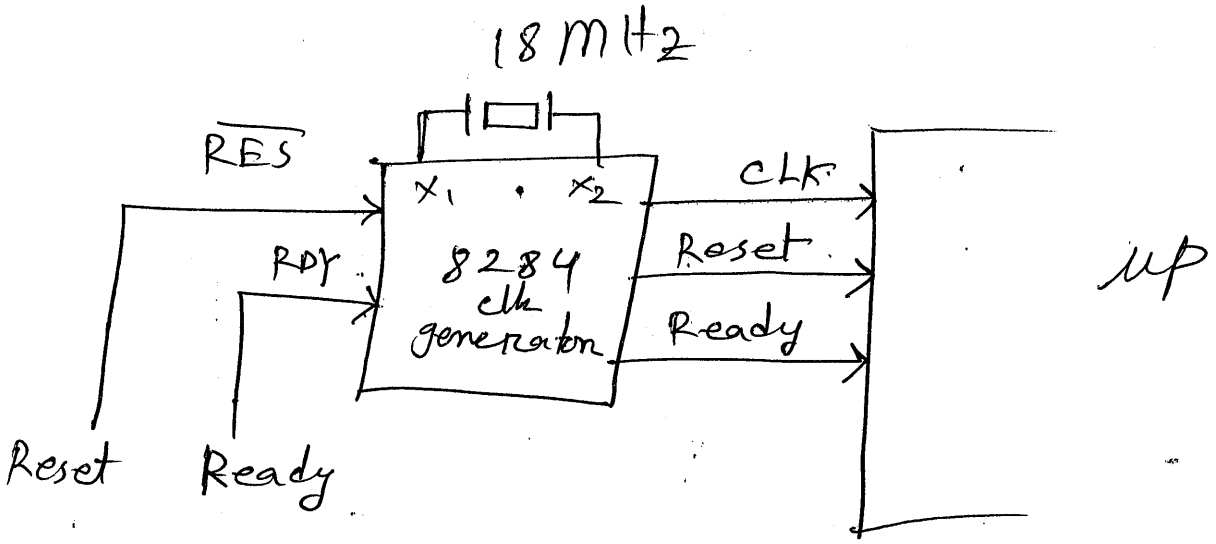
generate clock signal:

purpose is to generate 6MHz at 33% duty cycle.

↓

transition happens at mid point - 50% duty cycle

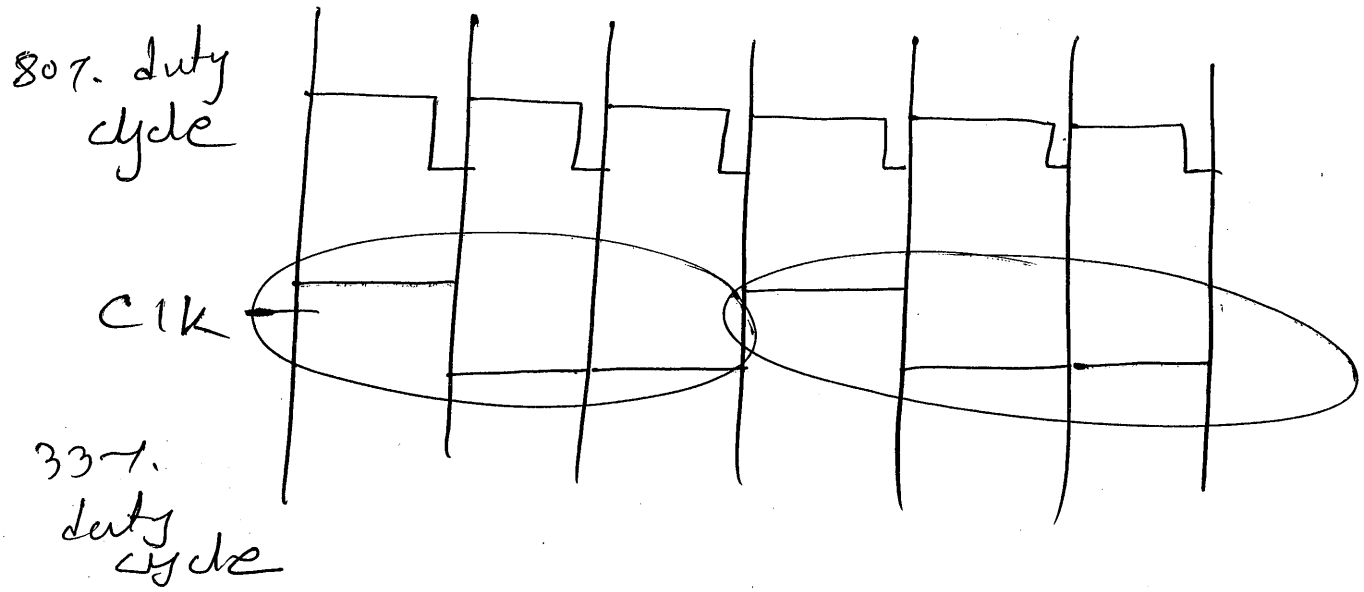
at 1/3rd point - 33% d.c.



* Reset signal provided to 8284, a synch reset signal sent to up, resets the up.

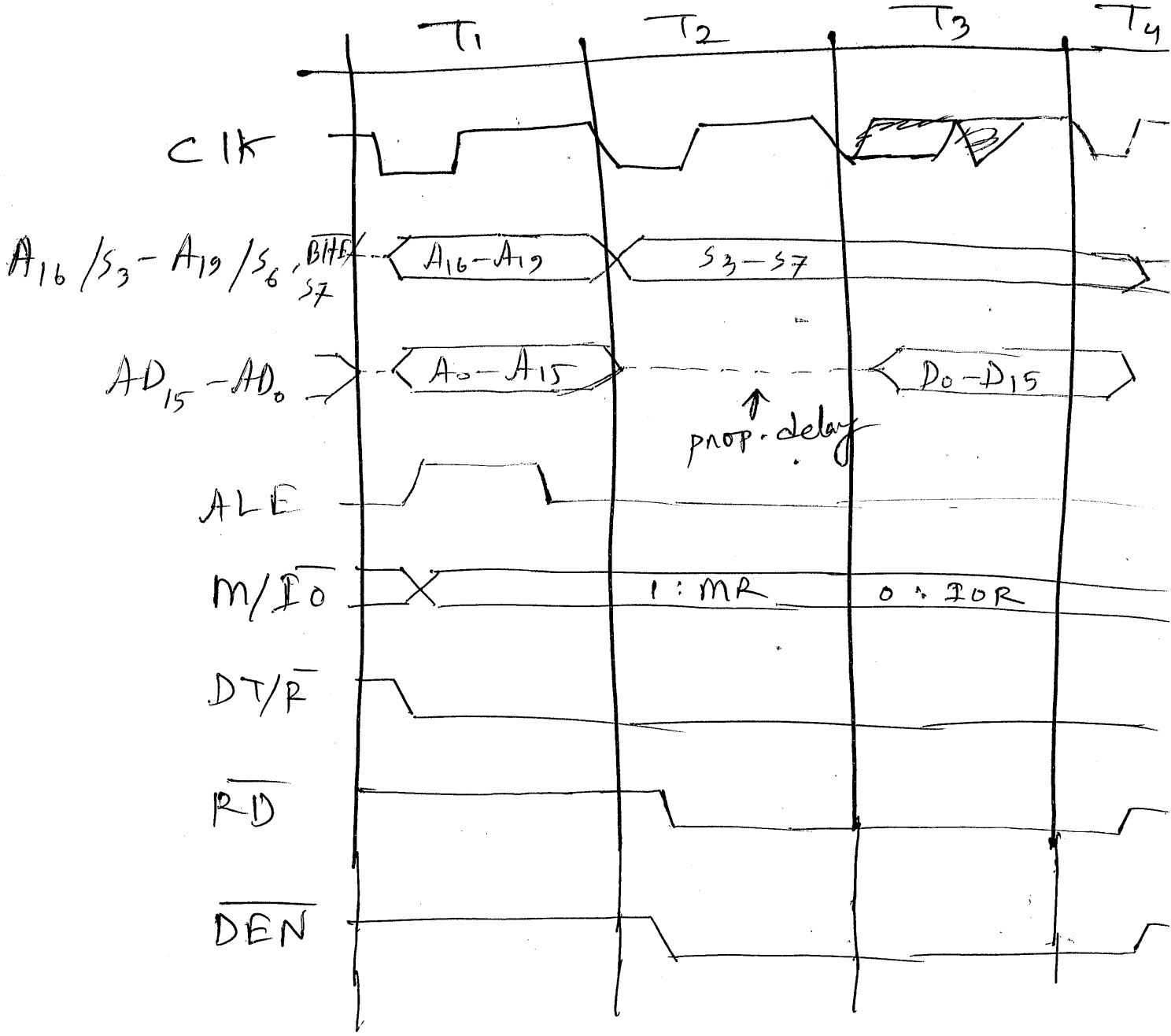
* Ready pin used to synch. the up with slow devices. if Ready = 1, the device is ready, if 0, device not ready, up waits for the device to be ready.
the device gives Ready signal to 8284, that sends a synchronised Ready signal to up.

* 8284, generates a 33% duty cycle from a random duty cycle. doesn't just divide by 3.



we produce $\frac{1}{3}$ pulse from 3 pulses.

Min mode read cycle



for write timing diagram,

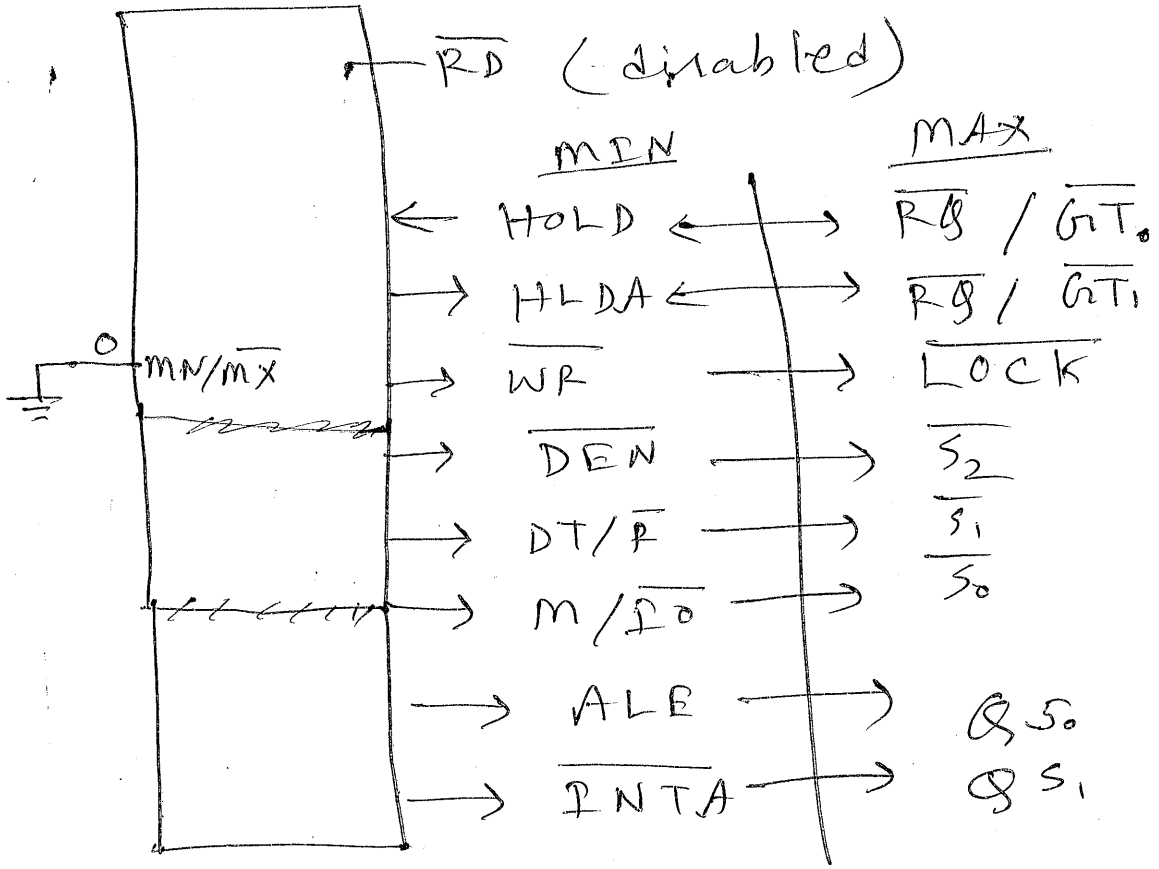
changes → $\overline{RD}$ → $\overline{WR}$

DT/$\overline{R}$ → high

$AD_{15} - AD_0$ → A0 - A15 | D0 - D15

↑ No prop. delay

Max. Mode

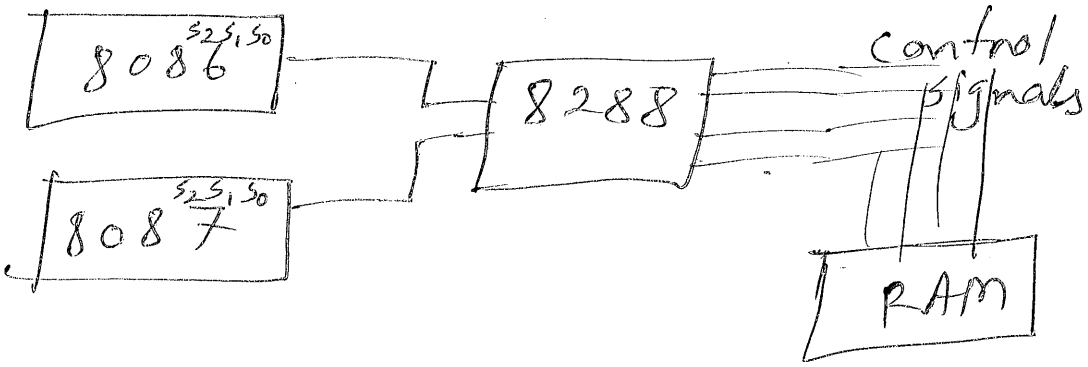


(A) • $\overline{RQ}/\overline{GT_0}$: 8087 wants to become BM, sends request by $\overline{RQ} = 0$, then 8086 sends grants by setting $\overline{GT_0} = 0$.

(A) $\overline{RQ}/\overline{GT_1}$: for another up. So, 8086 can have 2 ups with it to transfer bus controls with.

④ $\overline{LOCK} = 0$: buses will be locked until current instruction finished
- No intrpt allowed
- No HOLD granted

⑤ In Max mode, no up can generate control signals cause all up's control signals will have to connected to RAM and ckt. will be very complicated. to make things simpler, a bus controller 8288 is used.



how does 8288 generate control signals?

ans:

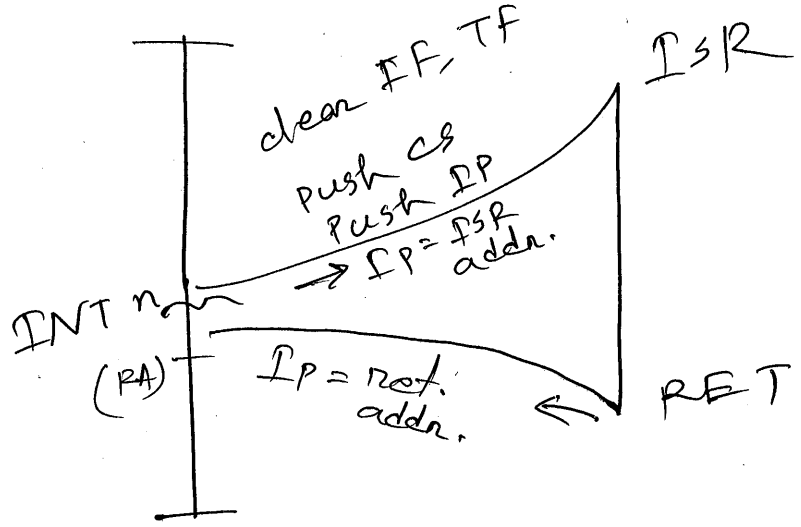
$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	
0	0	0	INTA
0	0	1	IO Read
0	1	0	IO write
0	1	1	Halt
1	0	0	Instn. fetch
1	0	1	Mem. read
1	1	0	Mem. write
1	1	1	Idle

(A)

QS1	QS0	prefetch q. status
0	0	No op
0	1	1st byte taken from queue
1	0	queue empty
1	1	subsequent byte taken

Interrupt

(B)



n = 0 ... 255 } 256 interrupts in 8086

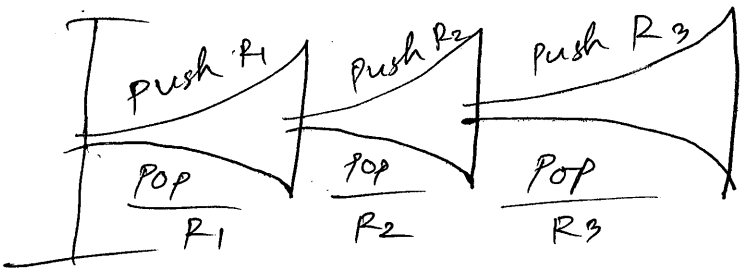
(C) when INT n is called, IP gets the value of corresponding ISR addr, then ISR is executed, while returning ip must get

the value from where it started ISR (basically the next instruction).

so, based on INT n, IP gets a fixed ISR addr, but returning isn't that rigid, it's flexible. IP can get any value depending on where it started.

* While "INT n" is being executed, IP already holds the addr. of next instn. which we need after coming back from ISR. so, we push IP to stack before loading ISR addr to IP and after returning POP IP from stack.

* Nested intrpts :

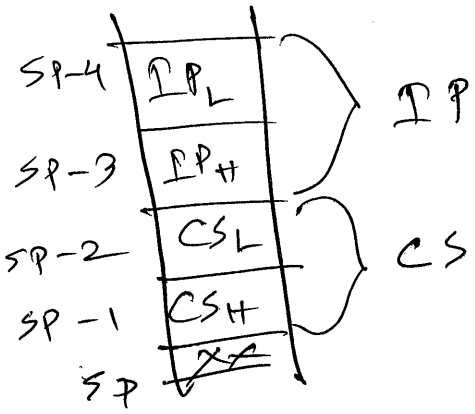


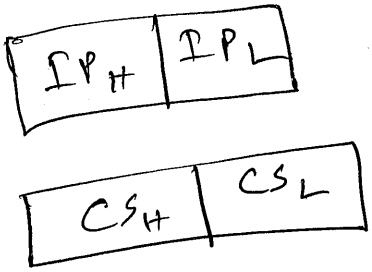
* During the ISR execution, don't mess with the stack i.e. # of push = # of pop must

* Push CS also if ISR is in different code segment from the main prog.

④ If you push cs and IP both, what to push first?

⇒





first push cs because cs is code segment addr, IP is just an addr. changing CS → changes your segment changing IP → " instr. so, CS is higher. (little Endian)

④ Every interrupt has a unique ISR addr. up should know it. How?

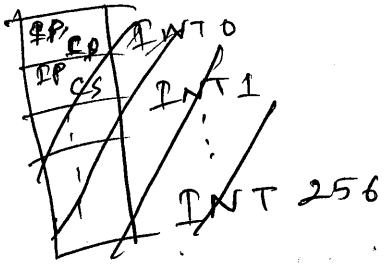
Int. Vector Table

IVT doesn't contain ISR ~~addr~~, it contains addr. of ISR. Where is IVT? in memory. Where is ISR? in memory. so, after getting an intrpt, up goes to memory twice. first to IVT to ~~get~~ fetch the ISR addr., then loads it to IP and finally goes to that addr.

in mem. again. So, IVT vectors the up or directs up to the correct ISR addr. hence the name Int. vector Table.

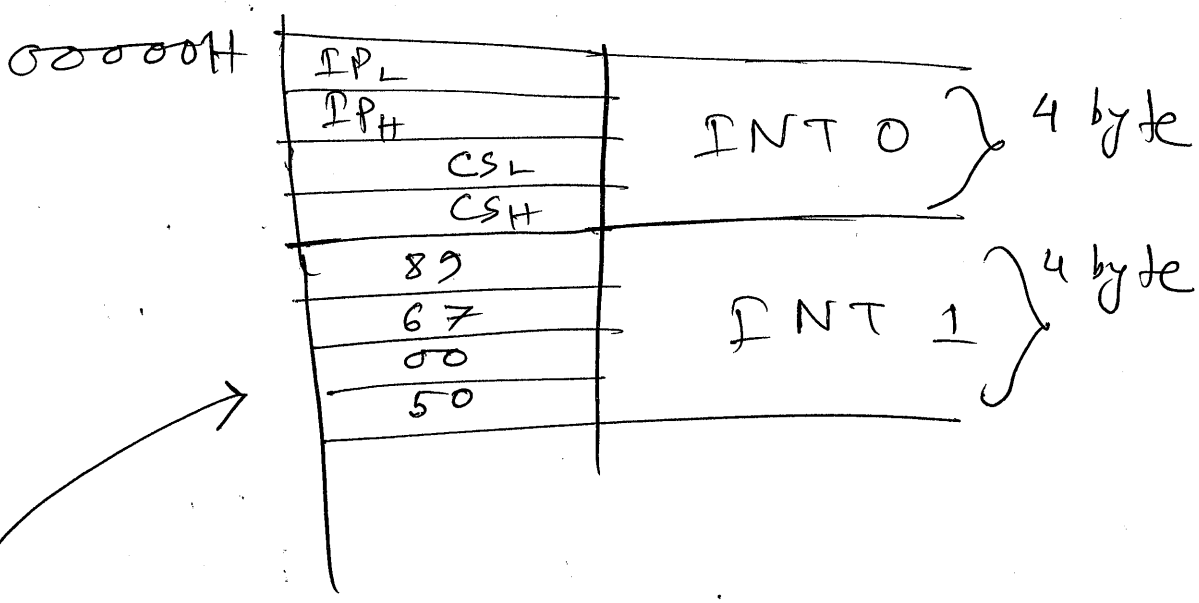
① Size of IVT :

every interrupt is ISR addr = CS and IP
= 16 bit + 16 bit
= 32 bit
= 4 byte



So, IVT sz = 256 × 4 bytes = 1024 bytes
= 1 KB

ex:



Say, INT 1 is ISR address is given by 56789

⇒ CS = 5000 _ IP = 6789

② INT n ⇒ IP : n × 4 : IP_L
n × 4 + 1 : IP_H
CS : n × 4 + 2 : CS_L
n × 4 + 3 : CS_H

④ clear IF, TF.
(0) (0)

④

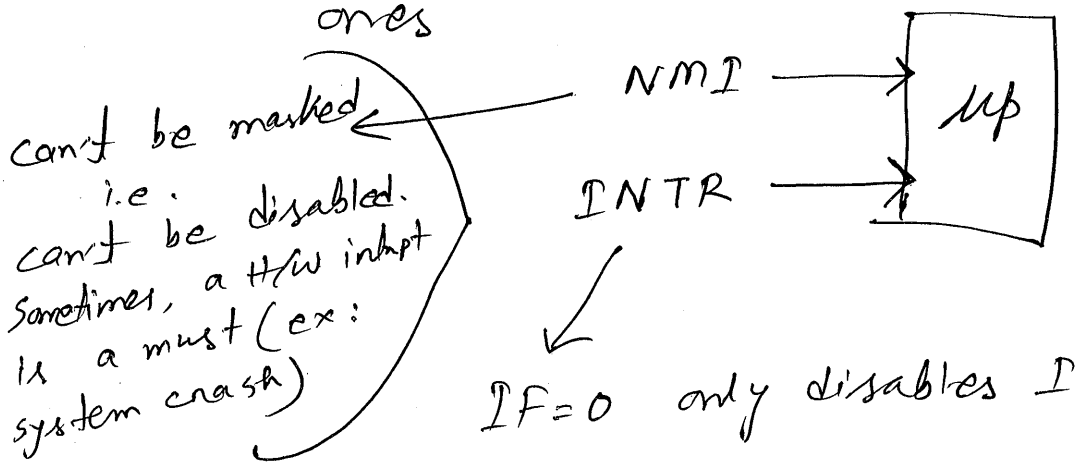
clear IF cause at a time wants to do one ISR.

for nested ISR, programmer has to manually set IF=1 by using the command STI.

- by IF=0, all 256 interrupts aren't disabled.

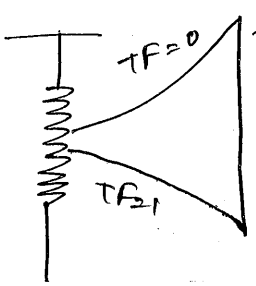
Interrupt
- H/W intrpt: external device can cause it to up.
- S/W intrpt: Interrupt is called by programmer by "INT n" command

- IF=0 sets H/W interrupts disabled.
- there are 2 H/W intrpts among 256 ones



n*4 is not the ISR address. It's the location from where up will fetch the ISR address

④ TF=0



- execute ISR faster
- coming back from ISR, TF=1 cause intrpt

shouldn't change the way program works

④ clear IF, TF.
(0) (0)

④

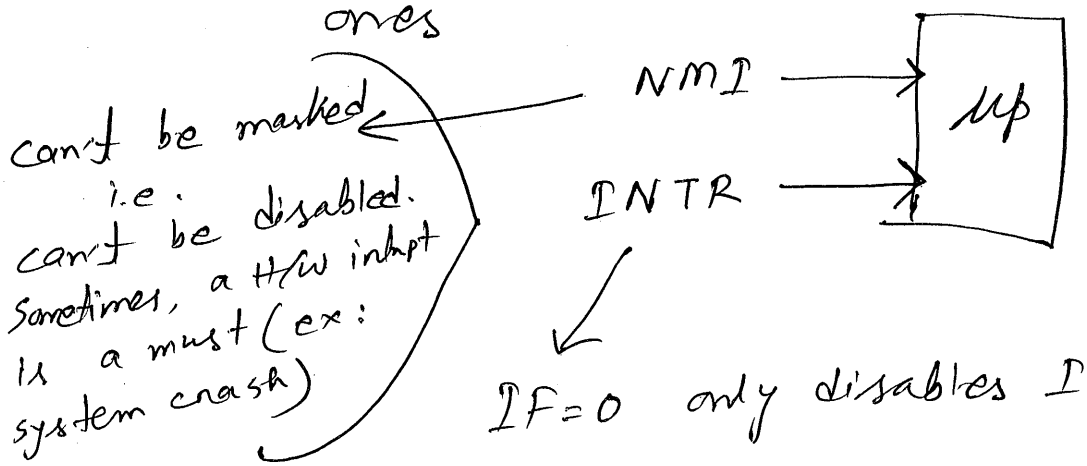
clear IF cause at a time wants to do one ISR.

for nested ISR, programmer has to manually set IF=1 by using the command STI.

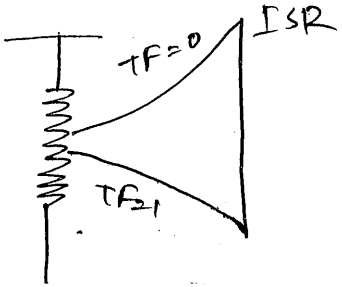
- by IF=0, all 256 interrupts aren't disabled.

Interrupt
- H/W intrpt: external device can cause it to up.
- S/W intrpt: Interrupt is called by programmer by "INT n" command

- IF=0 sets H/W interrupts disabled.
- there are 2 H/W intrpts among 256 ones



④ TF=0



- execute ISR faster
- coming back from ISR, TF=1 cause intrpt

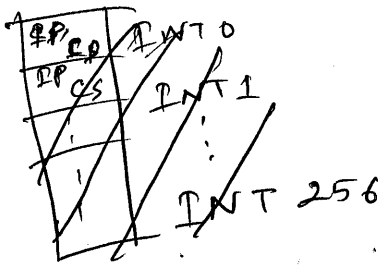
shouldn't change the way program works

n*4 is not the ISR address. It's the location from where up will fetch the ISR address

in mem. again. So, IVT vectors the up or directs up to the correct ISR addr. hence the name Int. vector Table.

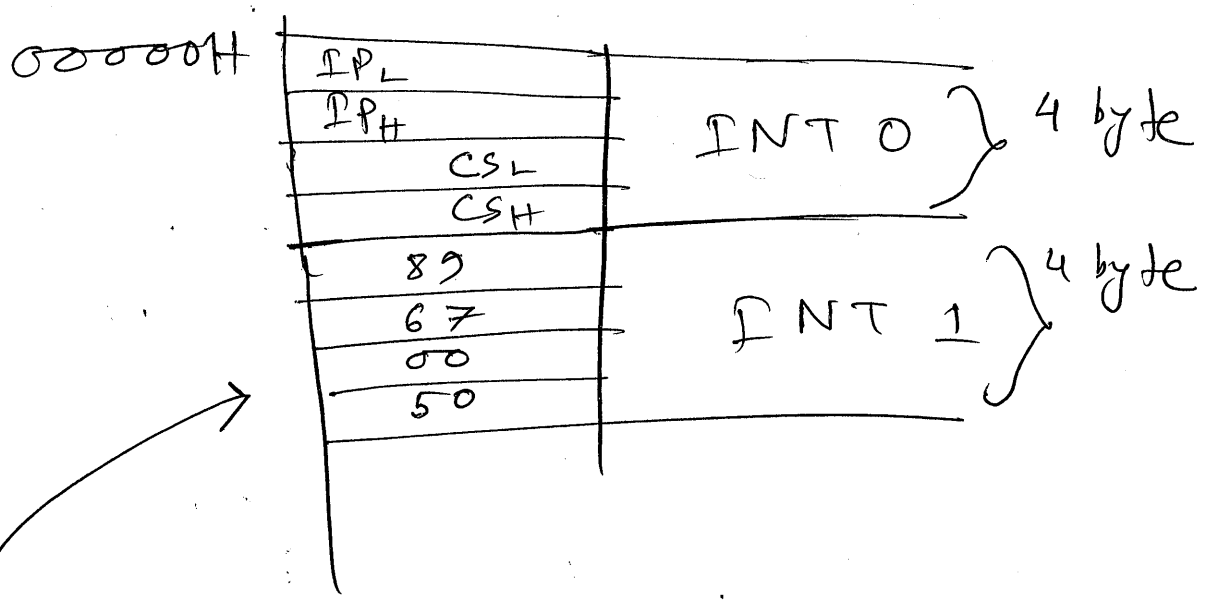
① Size of IVT:

every interrupt's ISR addr = CS and IP
= 16 bit + 16 bit
= 32 bit
= 4 byte



So, IVT sz = 256 × 4 bytes = 1024 bytes
= 1 KB

ex:



Say, INT 1's ISR address is given by 56789

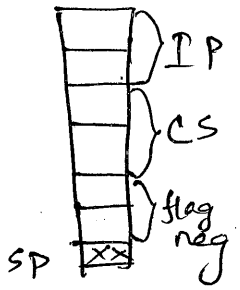
⇒ CS = 5000, IP = 6789

② INT n ⇒ IP: n×4 : IP_L
n×4 + 1 : IP_H
CS: n×4 + 2 : CS_L
n×4 + 3 : CS_H

- but, what if TF = 0 in our main program? Will we set it to 1 after coming back from ISR?
- NO, cause intrpt don't change anything of the program. We will push the entire flag register to stack, then make IF = TF = 0, come back from ISR, pop the flag register and restore its original values

So, in summary

```
push flag register
push cs
push IP
 :
 ISR
 :
pop IP
pop cs
pop flag register
```



(*) RET vs IRET

- return from ~~any~~ subroutine
- pops only CS, IP

- return from ISR
- pops flags, CS, IP

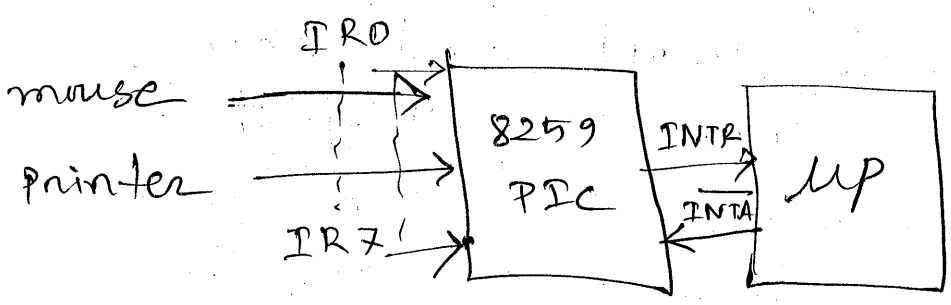
IVT

Int 0	Divide Error
Int 1	Single Stepping
INT 2	NMI
INT 3	Breakpoint
INT 4	Overflow

(*) If NMI gets an intrpt, we execute INT 2

If INTR _____, any intrpt can occur. The device who is triggering INTR, must specify a

vector number. Multiple devices can cause INTR to µp. So we need a device



µp sends $\overline{INTA}$ and asks for a vector number then PIC will send a n thru a 8 bit bus.

(*) Job of 8259 PIC :- accept intrpt on behalf of µp

(*) 8259 can take 8 intrpts → then resolve priority IR0 > IR1 > ..... IR7 → Provides vector number to µp.

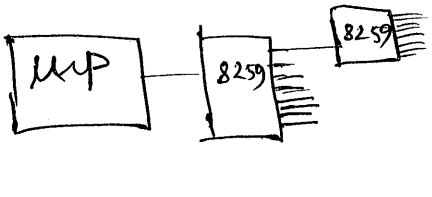
(*) vector numbers are initialized by programmers.

ex: IR0 → 40H
IR1 → 41H
⋮
IR7 → 47H
} Vector numbers

you set just IR0 then IR1 → IR7 is calculated

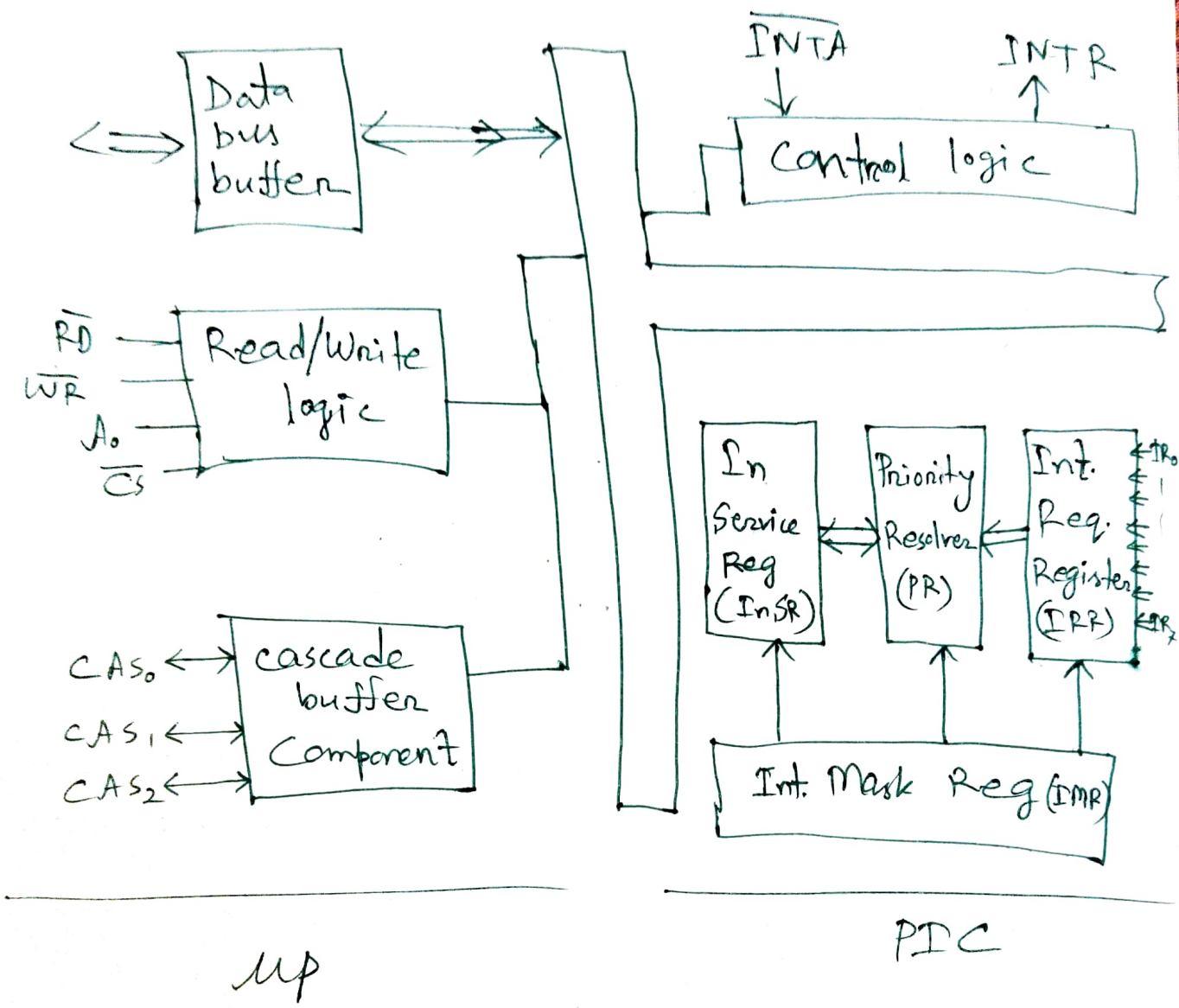
(*) Want more than 8 intrpts?

⇒ Cascade



} 64 intrpts

8259 - PIC



(A) IRR, InSR, IMR - 8 bit register

IRR - tells u which intrpt has occurred

IRR = 00000100 means INT 2 has occurred

InSR - tells u which intrpt is in service

IMR - tells u which intrpt is masked

④ InSR bit remains 1 when intrpt is being executed, after finished, it becomes zero.

⑤ Why these registors? We can see the values of $IR_1 - IR_7$ and send intrpt to up 8?

=> Say IR1 — ⌐¬ and after that IR2 — you send intrpt to up for IR1. and when it's finished, you come back and see IR2 ⌐¬ (the device sending IR2 might not keep high signal that long). So, you lose the IR2.

⑥ Say InSR = 00001000 i.e. INT3 is currently executing

at this time, IRR becomes 00000001

Since INT0 is of higher prionity than INT 3, so, PIC will send INT0 to up, execute that ISR

and then execute ISR of INT 3.

But if IRR becomes 0100 0000, i.e., INT 6 has occurred, up wont be disturbed because INT 6 is of lower priority than INT 3.

All these things are done by PR.

Before taking decision, PR will look at IMR to know if any intrpt is disabled.

(A) PIC sending a intrpt to up doesn't mean it will be executed. When up sends $\overline{INTA}$ to PIC, then the InSR register is updated. So getting $\overline{INTA}$ from up is a big thing for 8259.

(B) $\overline{INTA}$:

8259 gets ack.

8259 sends vector number

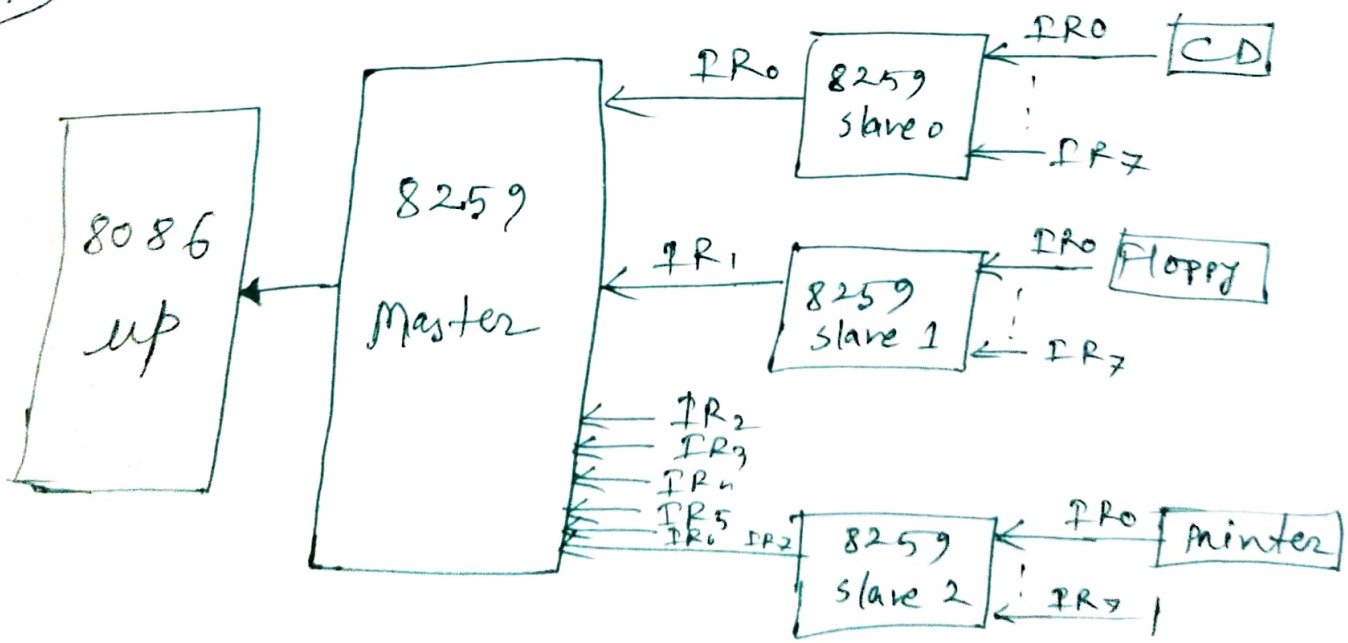
(C) After sending vector number to up, 8259 is disconnected from up. So,

its the job of up, i.e. job of programmer to inform 8259 that ISR has ended, you now make that InSR bit 0. This is done through the command EOI.

⊛ If EOI isn't given, and the INT 0 was being executed, 8259's InSR will have 1 at 0th position; therefore no other intrpt will occur ever !!!

Disaster

Ⓐ Cascaded Mode



In cascaded mode, every 8259 must be initialized.

When CD wants to intrpt up,
CD will intrpt slave 0
slave 0 will intrpt master
master ——— up
} 3 steps

up asks for vector number to master.
master asks ——— slaves.

But master won't ask for vector number if intrpt wasn't asked for by any slave, rather a device.

So, while initializing, master must be informed on which lines, there are slave.

(A) up gives $\overline{INTA}$ to master as well as all slaves. in the first $\overline{INTA}$ cycle. Before the next cycle, master sends CAS signals to all slaves and thus the slave chosen by master gets to know that it has to provide

vector number.

Then, when 2nd $\overline{INTA}$ signal goes to all slaves, only the slave chosen by the master will give vector number.

This is how prioirty will be resolved when both CD & printer sends intrpt (both IR0 of slave 0 and slave 1)

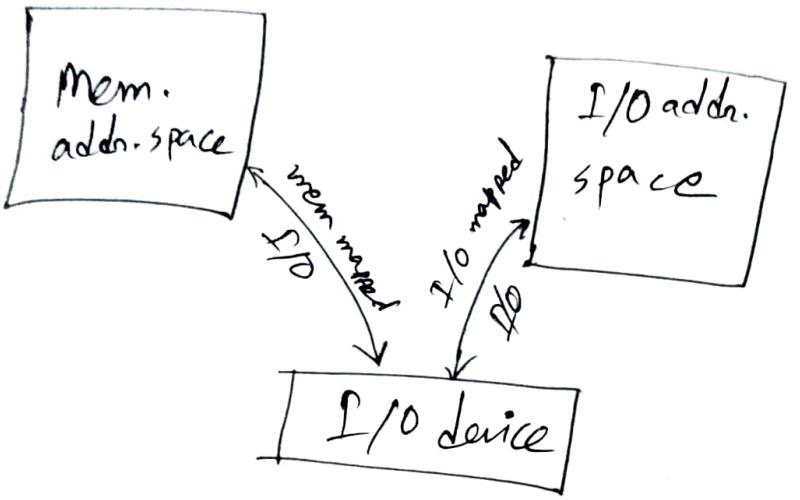
(A) CAS signals go from master to slave. But why they're bi-directional?

=> cause slaves are also 8259.

I/O

Address space < memory space (20 bit) 00000H ... FFFFFH

I/O space 0000H ... FFFFH (16 bit)



- treat an IO device like a memory device
- IO device gets a memory addr (20 bit)
- IN IO device
- up doesn't differentiate IO device from memory
- All mem. instruction
- All reg.
- only 2 control signals

- IO device mapped to I/O space
- I/O device gets I/O addr.
- 16 bit addr.
- 2^{16} = 64k IO devices
- up differentiates betn mem. and IO i.e. up separately treats memory and IO devices
- can only use IN, OUT
- A reg
- 4 control signals

- The main advantage of Mem Mapped IO is the # of IO devices can be up to 1M, much larger than 64K.

  cases where we need ≥ 64K IO devices
  - Burz khalifa - LEDs
  - Stadium - fire crackers

- Microprocessors always use IO Mapped IO.
- Microcontrollers use Memory mapped IO, cause they're used in industrial application
- up → sets $A_{15} - A_{16} = 0$ for IO address.

Advantages of IO Mapped IO

- Less decoding H/W
- Available mem. more

Advantages of Mem. Mapped IO

- less complex circuit
- Same instn. for both Mem & IO

Data transfer techniques

- Programmed IO
  - check ready states of IO device
  - If not ready, wait
  - when ready, start data transfer
  - wastes CPU time
- Intrpt. driven IO
  - Instead of waiting for IO, up does its work.
  - IO device, when ready, gives intrpt to up
  - data transfer initiated
  - CPU perf. improves

- DMA
  - The problem of tying up CPU for data transfer's duration is resolved
  - DMAC takes the charge

DMAC - 8237/ 8257

↓

from slide.

80286

- 16 bit data bus
- 24 bit addr. bus
- provides memory management & protection
- two modes - Real & virtual Protected

(*) Execution Unit contains registers → similar to 8086.
A special register MSW (Machine Status Word). The lower four bits are used. 1 bit → decides real/protected
3 bits → control coprocessor interface

(*) In real mode, A20 - A23 are disabled.
So, $A_0 - A_{19}$ → 1MB memory.
In protected mode, $A_0 - A_{23}$ all used
So, 16 MB memory.

Flag register

2 new pins

IOPL : Input Output Privilege Level flags

IOPL → 2 bits 00 is highest privilege

If current privilege level is higher than the IOPL, I/O executed without any hindrance. If IOPL higher than the current privilege, an interrupt occurs causing execution to suspend

NT : Nested task flag

When set, indicates that a system task has invoked another through a CALL instruction.

msw Register

- SMSW cx ; store msw into a reg.
- OR cx, 1 ; set the PE bit
- LMSW cx ; Load the new values back to msw

↳ enters into protected mode

⊛ The memory management, protection & task switching capabilities of 80286 are used by a multitasking OS to implement a microcomputer.

Memory management

real mode: similar to 8086

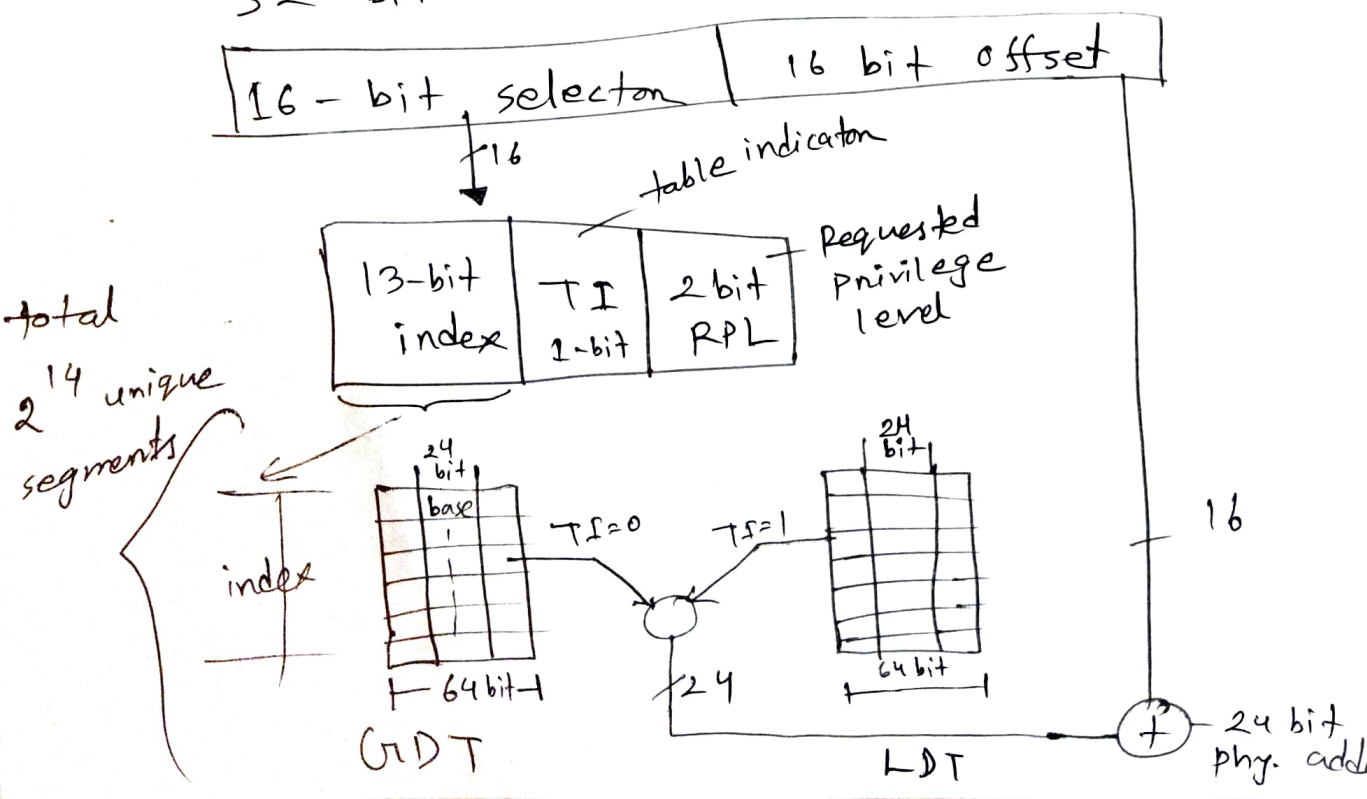
shift 16bit segment address 4 bit left

add to 16 bit offset

get 20 bit physical addr.

VPIM: In protected mode,

32 bit virtual address



each description table can store a maximum of 2^{13} descriptors.
each description - 2^3 bytes.
~~each segment~~ description can specify ~~upto~~ ~~2^{16} bytes~~. a segment of ~~2^{16} bytes~~.
Therefore a task can have its own address space upto $2^{13} \times 2^{16} = 2^{29}$ bytes. and can share 2^{29} bytes with all other tasks. Therefore, an address space of 2^{30} bytes can be assigned to a task (1 GB). So, virtual addr. space in 80286 - 1 GB

- since offset 16-bits, each segment is of 64k bytes
- A description contains info about a segment i.e. its base address, length, access rights; contains some other info too

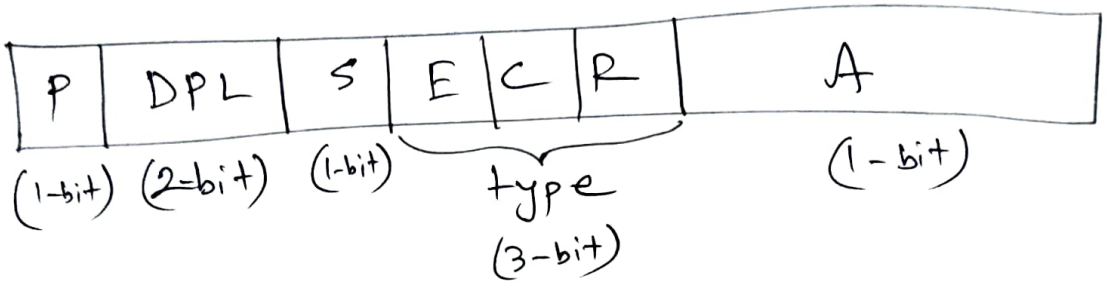
base addr : starting location of memory segment (24-bit)

length / limit :

length of segment − 1

16-bit allows segment upto 64kB maxm.

Access Rights Byte (8-bit)



P = 1 : segment is mapped to physical memory

P = 0 : No mapping to physical memory, hence, no base & limit.

DPL : sets the descriptor privilege level

S = 0 : system descripton

S = 1 : code/data "

type :
- 000 : data, read only
- 001 : data, read/write
- 010 : stack, read only
- 011 : stack read write

100 : code, execute only

101 : code, execute/read

110 : code, execute only

111 : code, execute/read

when CPL ≥ DPL

executable — conforming — readable

E C R

type

A = 0 : segment not accessed

A = 1 : segment has been accessed